REST Architecture

Representational state transfer (REST) originally specified in the 5th chapter of Roy Fielding's PhD thesis "Architectural Styles and the Design of Network-based Software Architectures" which specifies 6 constraints for the REST architecture.

1. Client-Server - API is separated from a client. Although this could represent different processes within a single computer. The objective is to allow for scalability- mulitple clients and multiple types of clients can thus connect with a single API.
2. Stateless - No cookies and no sessions. Clients must authenticate upon every request.
3. Cache - Server should provide caching directives so that in the event that a similar request arrives multiple times, a response can be provided without requiring a repeated request to the server. There are a variety of caches, including inside of the browser.
4. Layered System - Both the client and the API may or may not be communicating directly with the other. Likely there's a layer in between (such as a load balancer). This prevents a server from identifying requests by the client, in the case of a load-balancer, a server may receive *ALL* of its requests from a single client. The benefit of this is that in the event that a load balancer is placed in between the server and the client, you now have the ability to scale to a huge number of clients and servers without requiring fundamental changes.
5. Code-On-Demand - In practice this is an optional REST principle. Clients can receive executable code to run as a response to a request. How would an API know what types of code that a client could run?
6. Uniform Interface -
    i. Identification of resources - Resources represent every entity in the domain of the application. Each resource gets its own unique identifier URI such as: `/img/api/v1.0/images/3`. Collections of resources can also have identifiers such as `/img/api/v1.0/images`.
    ii. Resource Representations - Clients only access representations of a resource, never the resource itself. Additionally, clients only operate on representations of resources. The server may provide a number of different content types, like JSON or XML.
    iii. Self-Descriptive Message - Clients send and receive HTTP. Request method defines the operation; target is represented in the URI of the request, headers provide authentication credentials and content types, the representation of the resource is in the body, and the result of an operation is in the response status code.
    iv. HATEOAS Hypermedia As The Engine Of Application State - Clients don't know any resource URIs except for the root of the API and everything else can be discovered through links in resource representations.

REST defines an architectural style used in web development. Because it defined an architecture and not a specification, there is room for interpretation.

What I built

Can you teach a computer to recognize the brand logos of Nike and Altra in an unlabeled feed of images from Instagram?

In a previous project, I created a Convolutional Neural Network (CNN) that can identify brand logos in untagged/ unlabeled photos from a social media feed. The model I used implemented a previously trained network and transfer learning.

For this project I wanted to build a REST API that allowed me to send requests to that neural network and store its results in JSON.

How my project meets REST constraints

1. Client-Server - Yes, the API is hosted on a server both physically and logically separated from the clients I'm using to consume it.
2. Stateless - Yes, outside of the root URI for the API clients must authenticate upon every request. Although as listed in the below improvements section, adding rate limiting would potentially remove this constraint. In a statelsss system how would you identify an un-authenticated user in order to limit their number of requests?
3. Cache - Yes, this is being met when the API is consumed in a browser. However, as discussed in the below improvement section, adding etags would improve the existing solution.
4. Layered System - Yes, Similar to Client-Server above, all requests are authenticated separately, and there are no systems in place that would attempt to identify a client merely by its requests.
5. Code-On-Demand - No, I do not meet this optional requirement of REST. One potential application that I could see this constraint being used for is to download and execute JavaScript in the browser.

6. Uniform Interface -

    i. Identification of resources - Yes, each resource gets its own unique identifier URI such as: `/img/api/v1.0/images/3` and each collections of resources has an udentifier `/img/api/v1.0/images`.
    ii. Resource Representations - Yes, clients can only operate on representations of resources. In this case, the server only provides JSON content.

    iii. Self-Descriptive Message - Yes, the target is represented by the URI of the request, headers provide authentication credentials and content types, the representation of the resource is in the body, and the result of an operation is in the response status code. Request methods define the operation being performed as follows:

        - GET requests get data
        - POST requests create new data
        - PUT requests update existing data - although there's a differing opinion that says you should use POST for everything.
        - DELETE requests delete data

iv. HATEOAS Hypermedia As The Engine Of Application State - Yes, Every resource request returns a URI property (except DELETE) that is the full URI of where a resource resides. Potential improvments for this feature are discussed below.

Versioning

There's some argument about *where* to place the version information for your API, but not *if* you should version it. I place versioning info in the URI. The other alternative is in the header.

Authentication

I use HTTP Basic Auth

JSON

It's ubiquitous. It's human readable. It's easy to work with.

References

http://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api Miguel Grinberg - Is Your REST API RESTful? - PyCon 2015

Unit Tests

- Unit tests are included in the file `test_app.py`
- Also, include tests that validate success.

Below is sample output from running the unit tests. Note that there is a deprecation warning from TensorFlow that is expected.

```
$ python test_app.py
.........W tensorflow/core/framework/op_def_util.cc:332] Op BatchNormWithGlobalNormalization
...
----------------------------------------------------------------------
Ran 13 tests in 4.186s

OK
```

3 endpoints

1. Images File Store

HTTP Method	URI	Action
GET	[hostname]/img/api/v1.0/imgs	Retrieve list of images
POST	[hostname]/img/api/v1.0/imgs	Create new image
GET	[hostname]/img/api/v1.0/imgs/[img_id]	Retrieve an individual image
PUT	[hostname]/img/api/v1.0/imgs/[img_id]	Update an existing image
DELETE	[hostname]/img/api/v1.0/imgs/[img_id]	Delete an existing image

GET [hostname]/img/api/v1.0/imgs

Results:

```
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 502
Server: Werkzeug/0.9.6 Python/2.7.12
Date: Wed, 24 Aug 2016 22:06:07 GMT

{
  "images": [
    {
      "resize": false,
      "results": "",
      "size": "",
      "title": "Nikes",
      "uri": "http://10la.pythonanywhere.com/img/api/v1.0/images/1",
      "url": "http://imgdirect.s3-website-us-west-2.amazonaws.com/nike.jpg"
    },
    {
```

```
      "resize": false,
      "results": "",
      "size": "",
      "title": "Altra",
      "uri": "http://10la.pythonanywhere.com/img/api/v1.0/images/2",
      "url": "http://imgdirect.s3-website-us-west-2.amazonaws.com/altra.jpg"
    }
  ]
}
```


POST [hostname]/img/api/v1.0/imgs

Results:


```
HTTP/1.0 201 CREATED
Content-Type: application/json
Content-Length: 233
Server: Werkzeug/0.9.6 Python/2.7.12
Date: Wed, 24 Aug 2016 22:13:57 GMT

{
  "image": {
    "resize": false,
    "results": "",
    "size": "",
    "title": "",
    "uri": "http://10la.pythonanywhere.com/img/api/v1.0/images/3",
    "url": "http://imgdirect.s3-website-us-west-2.amazonaws.com/neither.jpg"
  }
```


GET [hostname]/img/api/v1.0/imgs/[img_id]

Results:


```
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 186
Server: Werkzeug/0.9.6 Python/2.7.12
Date: Wed, 24 Aug 2016 22:19:02 GMT

{
```

```
  "img": {
    "id": 3,
    "resize": false,
    "results": "",
    "size": "",
    "title": "",
    "url": "http://imgdirect.s3-website-us-west-2.amazonaws.com/neither.jpg"
  }
```

```
        |
```

PUT [hostname]/img/api/v1.0/imgs/[img_id]

```

```

Results:

```
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 195
Server: Werkzeug/0.9.6 Python/2.7.12
Date: Wed, 24 Aug 2016 22:43:20 GMT

{
  "img": {
    "id": 3,
    "resize": false,
    "results": "",
    "size": "",
    "title": "C-ron-ron",
    "url": "http://imgdirect.s3-website-us-west-2.amazonaws.com/neither.jpg"
  }
```

DELETE [hostname]/img/api/v1.0/imgs/[img_id]

```

```

Results:

```
    HTTP/1.0 200 OK
    Content-Type: application/json
    Content-Length: 20
    Server: Werkzeug/0.9.6 Python/2.7.12
```

```
Date: Wed, 24 Aug 2016 23:13:36 GMT

{
  "result": true
}
```

2. Image Inference

PUT [hostname]/img/api/v1.0/resize/[img_id]

```
```

Results:

```
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 415
Server: Werkzeug/0.9.6 Python/2.7.12
Date: Wed, 24 Aug 2016 22:47:30 GMT

{
  "img": {
    "id": 2,
    "resize": false,
    "results": {
      "results_name_1": "neither",
      "results_name_2": "altra",
      "results_name_3": "nike",
      "results_score_1": "\"0.7680\"",
      "results_score_2": "\"0.2004\"",
      "results_score_3": "\"0.0316\""
    },
    "size": "",
    "title": "Altra",
    "url": "http://imgdirect.s3-website-us-west-2.amazonaws.com/altra.jpg"
  }
}
```

3. Image Resize

```
```

Results:

```
HTTP/1.0 200 OK
Content-Type: application/json
Content-Length: 235
Server: Werkzeug/0.9.6 Python/2.7.12
Date: Wed, 24 Aug 2016 23:18:53 GMT

{
  "img": {
    "id": 2,
    "resize": false,
    "results": "",
    "size": {
      "height": 480,
      "width": 480
    },
    "title": "Altra",
    "url": "http://imgdirect.s3-website-us-west-2.amazonaws.com/altra.jpg"
  }
}
```

Improvements:

- better type checking for images
- better authentication beyond HTTP Basic
- HTTPS for every request
- Better HATEOAS compliance
- Improved caching via etags.
- add pagination
- add rate limiting
- add direct upload to AWS S3 with IAM roles